UNIVERSIDADE DE SÃO PAULO

Instituto de Ciências Matemáticas e de Computação

Especificação e implementação de um componente P2P para o middleware Ginga

Armando Biagioni Neto

São Carlos – SP

Especificação e implementação de um componente P2P para o middleware Ginga

Armando Biagioni Neto

Orientadora: *Maria da Graça Campos Pimentel*

Monografia de conclusão de curso apresentada ao Instituto de Ciências Matemáticas e de Computação – ICMC-USP – para obtenção do título de Bacharel em Informática.
Área de Concentração: Sistemas distribuídos

USP - São Carlos
Junho de 2010

Dedico este trabalho a minha família, patrocinadores da minha vida.

Agradecimentos

Dedico meus sinceros agradecimentos para todos aqueles que chegaram antes e criaram esta máquina mágica, o computador; agradeço também a todos aqueles que quando escutam a palavra amigo se recordam de mim.

Resumo

O objetivo deste trabalho é incluir no Ginga, *Middleware* do Sistema Brasileiro de TV Digital (SBTVD), um componente que possibilite a comunicação através de uma rede de arquitetura *Peer-to-Peer* (P2P) e ofereça uma Interface de Programação de Aplicativos, (*Application Programming Interface*), ou seja, uma camada de abstração para que programadores utilizem o recurso de troca de arquivos dentro do ecossistema do *middleware* Ginga sem se preocuparem com detalhes de implementação dos protocolos de comunicação Ponto-a-Ponto. Durante a elaboração deste trabalho foram estudados o funcionamento e a estrutura do *middleware* e as ferramentas de auxílio ao programador, além dos protocolos de comunicação que poderiam ser utilizados dentro do escopo de troca de arquivos e mensagem. O trabalho obteve, a partir do uso do protocolo *XMPP-Jingle*, uma versão inicial do componente, sua especificação e uma aplicação de exemplo para teste. Assim, este trabalho mostra a capacidade de componentização do middleware para o uso de aplicações não convencionais.

Palavras-chave: Televisão Digital, Peer-to-Peer, SBTVD, middleware Ginga, XMPP, Jingle

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas

API	*Application Programming Interface*
DHT	*Distributed Hash Table*
GNU	*GNU is Not Unix*
ISO	*International Organization for Standardization*
NCL	*Nested Context Language*
OSI	*Open System Interconnection*
P2P	*Peer-to-Peer*
SBTVD-T	*Sistema Brasileiro de TV Digital Terrestre*
XML	*eXtensible Markup Language*
XMPP	*eXtensible Messaging and Presence Protocol*

1 Introdução

1.1 Contextualização e Motivação

O sinal de televisão digital está sendo implementado em todo o território nacional, com isso, novos recursos são disponibilizados, entre eles a possibilidade de interação com os programas transmitidos, a partir de aplicações desenvolvidas para esse fim. A comunicação ponto-a-ponto entre os espectadores permite a colaboração e troca de arquivos, um novo recurso disponível a ser explorado pelos desenvolvedores das aplicações para TV Digital Interativa (TVDi).

O *Middleware* Ginga é um projeto realizado em conjunto com várias universidades e coordenado pelos laboratórios Telemídia da Pontifícia Universidade Católica do Rio de Janeiro (PUC-Rio) e o Laboratório de Aplicações de Vídeo Digital (LAViD) da Universidade Federal da Paraíba (UFPB). A Rede Nacional de Pesquisa (RNP) é a incubadora do programa Centro de Pesquisa e Desenvolvimento em Tecnologias Digitais para Informação e Comunicação (CTIC), a qual propôs uma série de tarefas para o desenvolvimento do *middleware* dentro do projeto intitulado: *GingaFrEvo e GingaRAP - Evolução do Middleware Ginga para Múltiplas Plataformas (Componentização) e Ferramentas para Desenvolvimento e Distribuição de Aplicações Declarativas*; o projeto é subdivido em GingaRAP, destinado ao suporte a autoria de aplicações, e GingaFrEvo, um *framework* de evolução da tecnologia Ginga.

Inserido no GingaFrEvo está o GingaAiyê, especialização do Ginga-CC (*Ginga-Common Core*) para aplicações não convencionais. Ginga-CC é o nome dado a um dos grandes módulos do *middleware* Ginga, o Núcleo Comum, que será abordado mais a frente neste trabalho. Uma das motivações para o GingaAiyê é a demanda pela padronização do *middleware* Ginga para plataformas de IPTV, Internet TV e P2P TV, abrindo a possibilidade para que os usuários realizem atividades colaborativas por meio da Internet. Com isso em mente uma das tarefas designadas ao Laboratório Intermídia foi a especificação e implementação de um componente P2P que proporcione a colaboração entre usuários via servidores apropriados (SOARES, 2009a), originando então este trabalho.

Nos computadores a troca de arquivo entre usuários é um elemento essencial para a

colaboração, seja por e-mail ou redes de compartilhamento, com atenção a esse potencial, o desenvolvimento da interatividade da TV Digital deve oferecer tal recurso, de modo que desenvolvedores terão mais uma importante ferramenta, que possibilitará o surgimento de aplicações mais interessantes para a televisão.

1.2 Objetivos

O objetivo do trabalho é obter ao final de três meses um componente escrito em C++, sua documentação, e uma aplicação de exemplo. Tal componente deve fornecer recursos de conexão e colaboração entre usuários, como troca de mensagens, dentro do contexto de TV digital para facilitar a criação de aplicações que tenham esse objetivo. Será fornecida como exemplo do uso do componente uma aplicação de teste para troca de arquivos.

1.3 Organização do Trabalho

Esta monografia está organizada de forma a refletir todos os passos realizados no desenvolvimento do projeto de graduação.

No Capítulo 2 é apresentada a revisão bibliográfica realizada para o trabalho, abordando o *middleware* Ginga, os potenciais protocolos utilizados e as ferramentas de desenvolvimento.

No Capítulo 3 são apresentados os métodos utilizados a partir do protocolo escolhido, Jingle, e o uso da biblioteca adequada, *libJingle*, e as ferramentas do sistema GNU de desenvolvimento.

Por fim, no Capítulo 4 são apresentadas conclusões a respeito do trabalho realizado.

2 Revisão Bibliográfica

Neste capítulo serão caracterizados os tópicos que sustentam o projeto: o Sistema Brasileiro de TV Digital, o *middleware* Ginga e o protocolo para comunicação ponto-a-ponto. Será então explicada a arquitetura do *middleware* Ginga e os protocolos que foram pesquisados pelo aluno para provável adoção, BitTorrent e XMPP-Jingle.

2.1 Sistema Brasileiro de TV Digital Terrestre

A transmissão do sinal de televisão historicamente tem sido analógica. O Brasil, a partir do Fórum Brasileiro de TV Digital Terrestre tem definido uma série de normas técnicas para a construção do Sistema Brasileiro de TV Digital. O sinal digital traz como principais vantagens, em relação a sinal analógico, imagem em alta definição, transmitindo até 1080 linhas, contra 480 do sinal convencional; som multi-canal, visto que hoje apenas 2 canais (direito e esquerdo) podem ser transmitidos; transmissão de múltiplos programas, já que uma mesma emissora pode transmitir simultaneamente vários programas nos chamados multicanais; e, além disso, o ponto de maior interesse para este trabalho, a existência de interatividade com a televisão.

Para decodificar o sinal transmitido é necessário que o televisor suporte o padrão de Transmissão Digital nacional, *Integrated Services Digital Broadcasting – Terrestrial Brazil* (ISDB-TB), ou adquirir separadamente um *Set-top-box*, nome dado ao dispositivo que ficará conectado a televisão e atuará como conversor do sinal. Esse dispositivo, fabricado por várias empresas, tem modelos mais simples, apenas com o decodificador do sinal digital; até modelos que possibilitam a interatividade e possuem o chamado canal de retorno, rede de dados como a internet, para enviar dados de volta a emissora.

As aplicações podem ser construídas em ambiente declarativo, usando uma linguagem de marcação; ou em ambiente imperativo, usando uma linguagem deste paradigma. O paradigma declarativo serve para descrever estruturas, em constraste, o paradigma imperativo descreve comandos a serem executados por procedimentos.

Para transparentemente disponibilizar às aplicações serviços de compressão, trans-

missão e modulação, define-se uma camada padronizada denominada *middleware* (MONTEZ; BECKER, 2006), a qual habilita a portabilidade das aplicações para diferentes receptores. A Figura 1 mostra as camadas de um *Set-top box*, onde aplicativos ficam separados em uma camada do *hardware* e Sistema Operacional. No SBTVD, esse *middleware* é o Ginga, obordado no próximo capítulo.
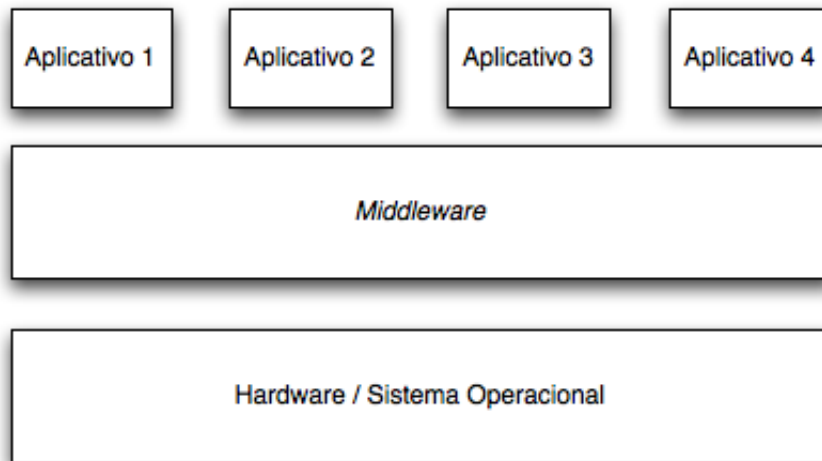
Figura 1: Camadas de um *Set-Top Box*

2.2 *Middleware* Ginga

A arquitetura do *middleware* Ginga, esquematizada na Figura 2, é formada por três grandes módulos, o ambiente declarativo Ginga-NCL, o ambiente imperativo Ginga-Imp e o núcleo comum Ginga-CC. O Núcleo Comum do *middleware* Ginga provê as funcionalidades comuns aos outros dois ambientes e é composto pelos decodificadores de conteúdo comuns e por procedimentos de obtenção de conteúdo dos fluxos de transporte, *Transport Streams* (TS), MPEG-2 e outras redes suportadas pelo sistema operacional do receptor onde o *middleware* Ginga está instalado. Além disso Ginga-CC deve suportar o modelo de exibição, controle de acesso condicional, gerenciamento de contexto, persistência de objetos de multimídia e gerenciamento de atualizações.

O ambiente declarativo é responsável pelo processamento dos documentos declarativos, tais documentos são especificados pela PUC-Rio numa linguagem denomindada NCL (*Nested Context Language*) para a descrição do conteúdo exibido na televisão (SOARES, 2009b) e interpretada pela ambiente Ginga-NCL.

O ambiente imperativo é responsável pelo processamento das aplicações em linguagem imperativa. No caso do SBTVD-T, a implementação deste ambiente é o Ginga-J para aplicações são especificadas usando a linguagem Java (SOARES, 2009a).

Aplicações de Televisão Digital Interativa (TVDi), podem ser puramente declarativas (escritas em NCL), puramente imperativas ou híbridas. O documento inicial aceito pelo *middleware* pode ser escrito para qualquer um dos ambientes citados. Aplicações podem ser recebidas das várias redes disponíveis, entre elas o carrossel de dados da emissora, fluxo contínuo de dados pelo protocolo do padrão MPEG-2-TS, ou aplicações residentes. Na Figura 2, essas aplicações aparecem na caixa intitulada *DTV Native Applications / Services*. Os componentes do Núcleo Comum, além de serem utilizados pelos ambientes declarativo e imperativo, podem ser utilizados por aplicações vindas de qualquer fonte supracitada.
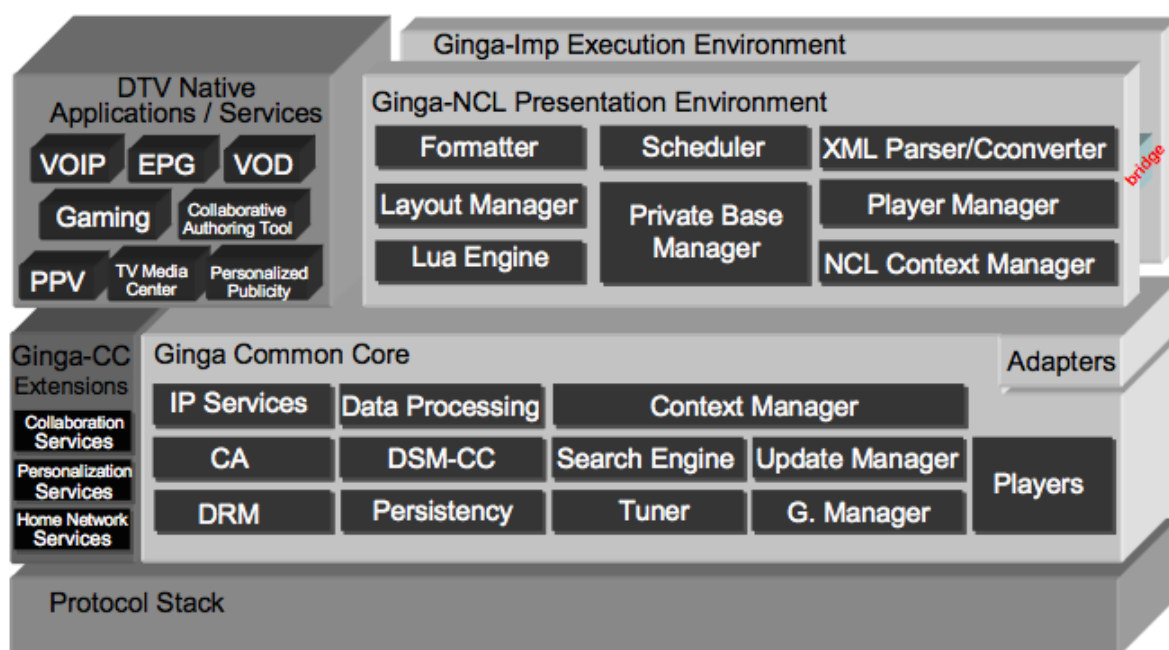
Figura 2: *Middleware* Ginga em um sistema cliente (SOARES, 2009a)

2.3 Sistemas Distribuídos *Peer-to-Peer*

Sistemas distribuídos são classificados em Cliente/Servidor e *Peer-to-Peer*. Um sistema Cliente/Servidor consiste em um sistema de alta performance, o servidor, e vários outros sistemas de performance mais baixa, os clientes. O servidor é o provedor de conteúdo e serviço. Um cliente requisita por conteúdo ou execução de serviços, sem compartilhar qualquer de seus recursos. Recursos podem ser poder de processamento, capacidade de armazenamento de dados, impressoras, entre outros; serviços são compartilhamento de arquivos, por exemplo.

Um sistema distribuído é considerado *Peer-to-Peer* se os membros compartilham parte dos seus recursos. Esses recursos são necessários para prover o serviço e conteúdo oferecido

pela rede e devem ser acessados pelos outros *peers* diretamente, sem passar por entidades intermediárias. Os participantes desse tipo de rede são provedores bem como requisitantes destes de serviços e conteúdos. Em P2P puro não há a existência de nenhuma entidade para prover recursos ao sistema distribuído (SCHOLLMEIER, 2001).

Nestre trabalho, em particular, os protocolos escolhidos para estudo de viabilidade e funcionalidade foram o BitTorrent, JXTA, Gnutella2, SIP e XMPP-Jingle. Serão abordados com maior profundidade BitTorrent e XMPP-Jingle, uma vez que foram os estudados pelo aluno, os outros protocolos ficaram sob responsabilidade de outros membros do laboratório.

2.3.1 Protocolo BitTorrent

BitTorrent é um protocolo destinado a distribuição de arquivos (COHEN, 2009), classificado como um sistema P2P híbrido, pois exige uma entidade chamada *tracker*, responsável por fornecer os endereços dos pontos da rede que disponibilizam o recurso (arquivo ou diretório) requisitado. O protocolo ainda permite que o sistema seja utilizado como P2P puro, uma vez que o *tracker* pode ser substituído por uma arquitetura de rede com Tabela *Hash* Distribuída (*Distributed Hash Table* - DHT), para a descoberta e troca dos endereços dos *peers*.

Um usuário final a fim de compartilhar um arquivo deve criar um *MetaInfo File* - arquivo com *mimetype application/x-bittorrent* e extensão *.torrent* - associado com o que deseja compartilhar, esse arquivo *.torrent* é codificado em *Bencode* — codificador desenvolvido pelo projeto BitTorrent — que possui implementação de referência escrita em *Python*[1]. A estrutura do arquivo *MetaInfo* é um dicionário com as possíveis chaves:

announce: URL do tracker;

info: outro dicionário mapeado com as chaves:

- **name:** sugestão para o nome do arquivo ou diretório a ser salvo
- **piece length:** número de bytes de cada pedaço que o arquivo vai ser dividido
- **pieces:** número de pedaços a dividir
- **length:** tamanho do arquivo
- **path:** lista com os nomes dos subdiretórios

Esse arquivo é distribuído através de um website, ou qualquer outro meio, onde outro usuário irá baixá-lo e, com seu cliente BitTorrent, requisitar do *tracker* os endereços dos

[1] *Bencode* - http://pypi.python.org/pypi/BitTorrent-bencode/5.0.8 — acesso em 27/05/2010

usuários que estão ofertando o arquivo desejado. O primeiro usuário a disponibilizar um arquivo é chamado de *Initial Seeder* (Semeador Inicial, em tradução livre) (COHEN, 2009).

Uma rede DHT pode ser entendida como uma tabela *hash*, onde cada nó da rede consegue fornecer um dado a partir de uma chave (GHODSI, 2006). A rede substitui o *tracker* para a requisição de endereços, sendo que cada nó da rede possui uma lista de endereços de seus vizinhos, assim cada nó conhece parte da rede, como em um grafo. Por exemplo, na Figura 3, caso o nó número 2 queira um arquivo que apenas o nó número 4 possui, ele requisitará aos seus vizinhos, nós 3 e 5, que também não possuem o arquivo e requisitarão ao nó 4, que por sua vez responderá positivamente a requisição. Com isso, o nó 4 se torna um vizinho do nó 2.
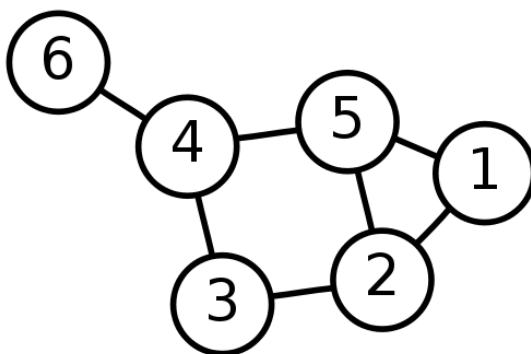
Figura 3: Exemplo de grafo numerado

A Figura 4 mostra uma rede distribuída. Os retângulos a esquerda representam os dados disponíveis, uma função *hash* é executada sobre esse dado, o resultado desta função é atribuído ao dado disponível. Cada *peer* responde positivamente quando um dado com *hash* igual a um dos arquivos disponíveis está sendo procurado. A nuvem a direita representa a rede, cada *peer* neste caso possui apenas um dado para fornecer. A função *hash* utilizada pelo BitTorrent é a *SHA1* que gera uma palavra de 160-bit.

Em uma rede DHT aplicada ao BitTorrent o *peer* é representado por um *node*, ou seja, seu endereço formado pelo IP e a porta do serviço DHT, esses *nodes* ao entrar na rede trocam dados com seus vizinhos, que estão em uma tabela de roteamento local, informando a disponibilidade dos arquivos para troca através do BitTorrent. Quando um cliente do protocolo começa a baixar um arquivo da rede requisita a seus conhecidos pelo arquivo, se os mesmos não tiverem o arquivo então perguntam a seus conhecidos e assim sucessivamente até que todos os *nodes*, que possam ser descobertos, sejam visitados. Essa dinâmica seria de grande valia, uma vez que não são necessários servidores, porém apenas clientes que possuem o endereço dos outros conseguem trocar endereços, ou seja, ainda é necessário um mecanismo para descoberta destes endereços por parte de nós que ainda não fazem parte da rede DHT.
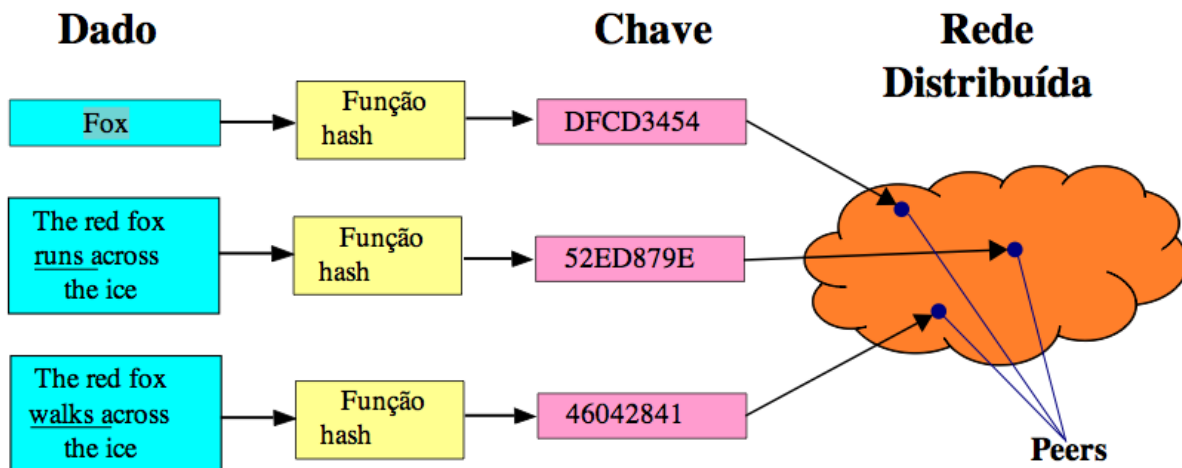
Figura 4: Tabela *Hash* Distribuída

A biblioteca usada, *libTorrent*, é distribuída sob licença BSD, escrita em C++ e *cross-plataform*, ou seja, capaz de ser compilada e utilizada em vários sistemas operacionais e acompanha um cliente de exemplo que foi usado para testar o mecanismo de troca de arquivos por BitTorrent.

As características do protocolo BitTorrent são eficientes para a troca de arquivos entre um número muito grande de usuários, mas os mecanismos acabam criando dificuldades para trocas de arquivos quando, por exemplo, um usuário quer enviar um arquivo a outro usuário diferente.

2.3.2 Protocolo XMPP-Jingle

Concebido em 1999, o protocolo extensível de Mensagens e Presença, *Extensible Messaging and Presence Protocol* (XMPP), era anteriormente conhecido por Jabber, nome da comunidade *open-source* que iniciou o projeto como um padrão aberto para comunicação em tempo real. O desenvolvimento do XMPP foi motivado pela carência de um protocolo aberto para troca de mensagens instantâneas.

Aplicativos XMPP são capazes de oferecer os serviços de criptografia de canal, autenticação, presença, lista de contatos, mensagens um-a-um ou em grupo, notificações, descoberta de serviço, controle de fluxo e seções multimídia.

Os serviços estão definidos nas especificações publicadas pela Força Tarefa de Engenharia de Internet[2] (*Internet Engineering Task Force* - IETF), e suas extensões nos Padrões Estabelecidos XMPP[3], da *XMPP Standards Foundation.*

[2]http://ietf.org/ — acesso em 25/05/2010
[3]http://xmpp.org/ — acesso em 25/05/2010

A extensão Jingle é usada para inicializar e administrar sessões multimídia entre duas entidades, de maneira interoperável com os padrões da Internet (LUDWIG et al., 2009). Devido a utilização do procotolo XMPP juntamente com a extensão Jingle, este trabalho refere-se a XMPP-Jingle sempre mencionando o uso de ambas implementações.

Stanzas são para XMMP o que o pacote é para a camada de rede do modelo OSI/ISO, ou seja, uma estrutura enviada entre os dois pontos (entidades) que estão se comunicando. Apresenta-se abaixo um exemplo de *Stanza* para carregar uma mensagem:

```
<message from="joao@gaggi.com/casa" to="maria@wooho.com/celular">
 <body>Estou bem hoje.</body>
</message>
```

2.3.2.1 Como XMPP funciona

A arquitetura da rede XMMP é descentralizada e baseada em cliente/servidor, ou seja, a comunicação não é feita diretamente entre os clientes, semelhante o que acontece no serviço de e-mail, conforme ilustrado na Figura 5. Quando um cliente de e-mail envia uma mensagem, a mesma é enviada ao servidor do destinatário, como um pacote de rede, a mensagem trafega entre os servidores de e-mail intermediários. O serviço de XMPP funciona de maneira diferente, os clientes se conectam aos servidores, que por sua vez se conectam entre si, como pode ser visto na Figura 6. Aqui os servidores onde os clientes estão conectados trocam dados diretamente, sem passar por nenhum outro servidor XMPP (*hop*) intermediário .

As contas dos usuários da rede XMPP são identificadas por um *JabberID* (JID) que possui a forma de um endereço de email, ou seja, *nomedousuario@servidor.tdl*, inclusive podendo ser o próprio endereço de e-mail, caso o servidor forneça ambos serviços. Um exemplo disto é o GMail da Google Inc. [4]

Um usuário pode se conectar à rede através de múltiplos dispositivos ou computadores concorrentemente, a cada conexão um *resource* é designado pelo servidor, ou definido pelo usuário, para servir de identificação e roteamento das mensagens. Dessa forma, quando, por exemplo, um usuário inicia através do celular uma conversa com um amigo, o mesmo responde e esta mensagem é encaminhada para o telefone, e não qualquer outro elemento conectado na rede, com *resource* diferente.

[4]Os serviços de DNS são usados pelo XMPP para os fins de resolução de domínios
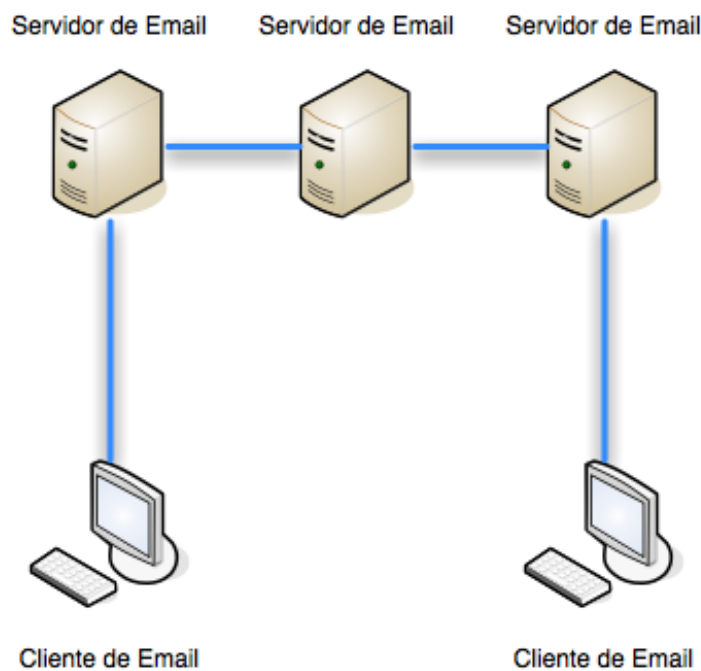
Figura 5: Arquitetura do E-Mail
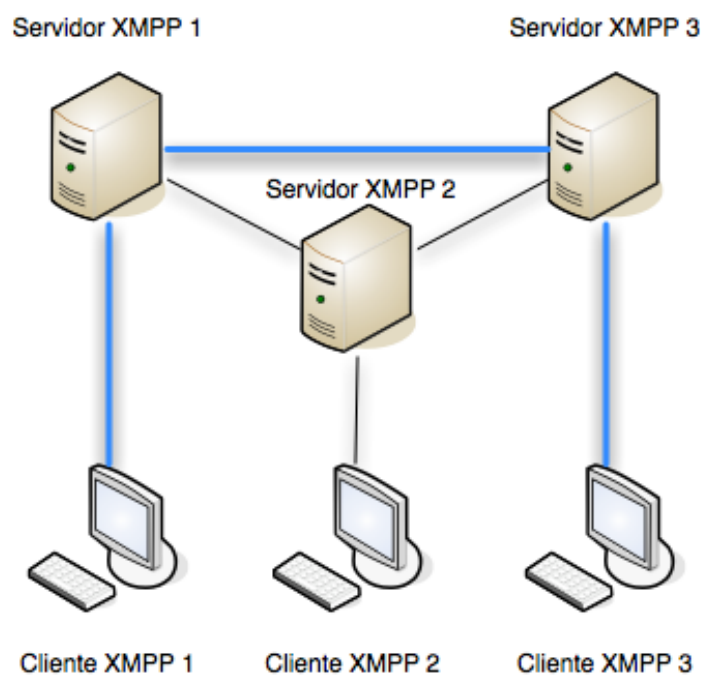
Figura 6: Arquitetura do XMPP

2.3.2.2 Biblioteca *libPurple*

Essa biblioteca suporta a maioria dos protocolos de mensagem instantânea. É a mais completa biblioteca de código aberto para esse fim, porém sua documentação é focada na comunidade de desenvolvedores que mantém a própria biblioteca e o cliente Pidgin,

escrito em GTK+. Por isso, a complexidade do código nos clientes-exemplo dificulta o entendimento no curto prazo em que este projeto deve ser executado. A *libPurple* implementa a extensão Jingle que dá suporte à troca de arquivos e sessões multimídia.

2.3.2.3 Biblioteca *libJingle*

Implementação do protocolo Jingle feita pela Google Inc., distribuída em código aberto. A especificação da extensão Jingle ainda está classificada como *draft* (não concluída) e portanto a *libJingle* não provê compatibilidade com outros sistemas que se utilizam da versão atual da especificação, uma vez que o desenvolvimento da biblioteca ocorreu paralelamente com as primeiras revisões da especificação.

2.3.2.4 Estrutura de aplicações com *libJingle*

A estrutura de um programa que se utiliza da biblioteca *libJingle* possui, além da interface com o usuário, três grandes componentes(GOOGLE, INC., 2009), mostrados no diagrama da Figura 7[5], que são:

***XMPP Messaging Component*:** Componente de mensagens XMPP, serve como um *gateway* entre a rede (representada no diagrama pela nuvem) e as *stanzas* que chegam do Componente de Lógica e Gerenciamento da Sessão;

***Session Logic and Management Component*:** Componente de Lógica e Gerenciamento da Sessão, cuida da lógica de cada tipo de sessão, é o responsável por passar *stanzas* para o componente Ponto-a-Ponto, que cuidará dos detalhes da conexão, e depois irá ler/escrever os dados do componente para arquivos, sistema de audio ou vídeo;

***Peer-to-Peer Component*:** Componente Ponto-a-Ponto, responsável por cuidar dos detalhes da conexão entre os pontos, alocando *sockets* e monitorando a qualidade da conexão.

Na Figura 7 o fluxo de dados (audio, vídeo, arquivos) está representado pela seta posicionada mais abaixo na ilustração, e o fluxo do XMPP pela seta que corta a figura pelo meio. O tráfego de dados é realizado Ponto-a-Ponto, diretamente entre os clientes que estão trocando arquivos, por exemplo. Assim o servidor XMPP é responsável pelo envio e recebimento das *stanzas* que anunciarão a troca P2P, mas os dados não passam pela rota do servidor.

[5]Disponível em `http://code.google.com/apis/talk/libjingle/libjingle_applications.html` — acesso em 31/05/2010

Uma aplicação, representada pela caixa *Application* no diagrama, deve explicitamente criar objetos da classe *SessionManager*, responsável pro criar e destruir objetos de *Session*, que cuidam do transporte dos dados; subclasse *PortAllocator*, responsável por alocar portas locais para o transporte do dados em P2P; subclasse *Session Client*, cuida das tarefas comuns a todas sessões; e uma classe que auxiliará no processo de *login*, cuidando da autenticação segura. Um processo de Sinalização é utilizado para a comunicação entre os objetos, aqui utilizando a biblioteca *sigslot*.
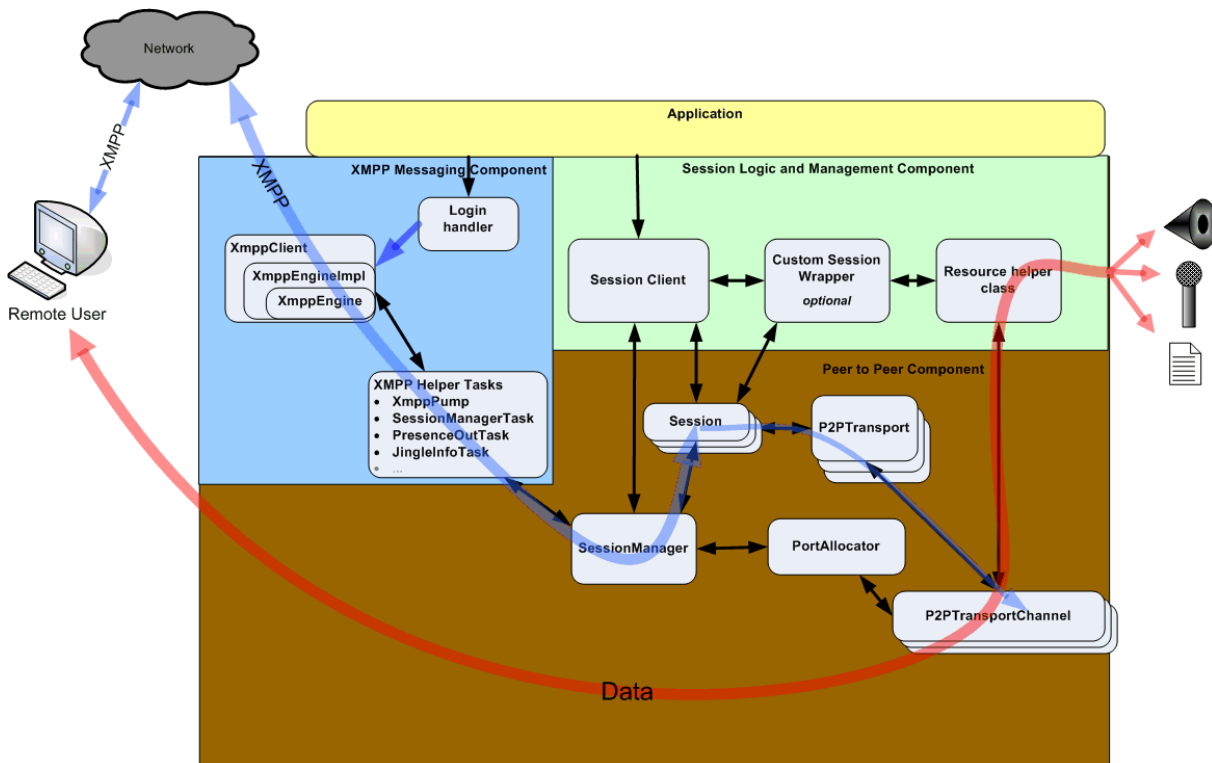
Figura 7: Diagrama da estrutura de um programa *libJingle* (GOOGLE, INC., 2009)

2.4 Considerações Finais

XMPP-Jingle foi o protocolo escolhido, uma vez que possuía as funcionalidades básicas desejadas para comunicação ou troca de arquivos. A biblioteca *libJingle* foi usada para a implementação final, uma vez que sua complexidade era bem menor comparada a *libPurple*. Além disso parte da preocupação residia no fato de construir um sistema distribuído com a maior liberdade em relação a servidores, pois dessa forma o programador de aplicações para televisão digital teria maior liberdade e menos com que se preocupar. A utilização do XMPP permite isso, mesmo dependendo de servidores por fornecer uma estrutura onde usuários já estão familiarizados, aplicações de mensagens instantâneas.

3 Desenvolvimento do Trabalho

Neste capítulo serão abordados o cronograma das atividades realizadas, bem como os detalhes das ferramentas utilizadas e decisões tomadas durante a execução do trabalho.

3.1 Projeto

O projeto consiste em implementar um componente P2P dentro do núcleo comum do *middleware* Ginga (Ginga-CC). Uma máquina virtual com a implementação de referência do *middleware* Ginga, e suas dependências, está disponível no repositório do Software Público[1]. A versão 0.11.2 foi utilizada para o desenvolvimento do componente.

Com o middleware em mãos e a definição do protocolo e biblioteca utilizados então o trabalho segue as seguintes etapas:

1. Configuração do ambiente de trabalho
2. Estudo da estrutura do código-fonte do *middleware* Ginga;
3. Estudo da biblioteca *libJingle*;
4. Estudo das ferramentas do *GNU Build System*;
5. Alteração da *libJingle* para geração de bibliotecas compartilhadas; e
6. Codificação do componente.

3.2 Descrição das Atividades Realizadas

Nesta seção estão descritas cada etapa da metodologia descrita na seção anterior, bem como todos os recursos, técnicas e sistemas utilizados em cada uma das etapas.

[1]http://softwarepublico.gov.br/dotlrn/clubs/ginga — acesso em 25/05/2010

3.2.1 Cronograma de Atividades Realizadas

O projeto teve início em 11 de Fevereiro de 2010, com a reunião que definiu a tarefa a ser executada e a data final para a entrega de uma primeira versão do componente rodando. A Tabela 1 mostra as atividades realizadas e datas onde marcos eram cumpridos.

Mês	Atividades Realizadas
Fevereiro	22/02: Reunião do grupo envolvido no projeto. Escolhas e sugestões de protocolos a serem estudados.
Março	Pesquisas dos protocolos que possibilitassem troca de arquivos em sistemas distribuídos P2P puros. 30/03: Reunião de discussão dos protocolos. Apresentação do Cliente torrrent rodando sem servidores. Outros membros do grupo apresentaram seus protocolos estudados.
Abril	Definição de que não poderia ser implementado um sistemas serverless. Grupo dividido para pesquisas como XMPP e SIP (Session Initiation Protocol). Trabalhos realizados em paralelo: implementação dos protocolos em exemplos com diversas bibliotecas.
Maio	06/05: Entrega do componente rodando e usado como referência desse trabalho. Revisão bibliográfica sobre sistemas distribuídos e redes P2P e contextualização dos sistemas de TV Digital e middleware Ginga. 31/05: Entrega desta monografia.

Tabela 1: Cronograma de Atividades

3.2.2 Configuração do ambiente de trabalho

Para o desenvolvimento foi utilizada uma máquina virtual para o *VMWare Player*, versão distribuída gratuitamente da ferramenta de virtualização *VMWare*, com a implementação de referência (em C++) do *middleware* Ginga e seu ambiente declarativo Ginga-NCL. Essa máquina virtual vem com a distribuição *Fedora Core 7* do sistema operacional *GNU/Linux*. A instalação desse sistema não possui gerenciadores de janela e, após a inicialização, o *framebuffer* é preenchido com um aviso introdutório com o mapeamento entre as teclas do teclado do computador e as teclas de interação do controle

remoto. A interação com o sistema para configuração e testes é feita acessando um serviço de terminal seguro (SSH), assim sendo todas as modificações no *middleware* Ginga foram submetidas a máquina virtual para testes. O programa de linha de comando `ginga` é o responsável por executar o middleware no sistema.

O código-fonte do *middleware* está em repositórios *Subversion* para controle de versão. Antes do desenvolvimento do componente, a última versão dos pacotes do *middleware* Ginga foram baixadas.

3.2.3 Estudo da estrutura do código-fonte do *middleware* Ginga

Como primeira tarefa foi necessário entender como estava estruturada a arquitetura do middleware, uma vez que sua documentação é escassa. Foi muito importante contar com a ajuda de membros do laboratório intermídia para essa tarefa. A versão utilizada no trabalho foi a 0.11.2 da implementação de referência.

O código-fonte do *middleware* está dividido em vários pacotes, sendo que *ginga-cpp* e *gingacc-cpp* são o foco do componente. O primeiro possui a implementação do 'main.cpp', que gera o binário 'ginga', programa de linha de comando que recebe como parâmetro de entrada um documento NCL. Já o segundo é formado por todos os subpacotes que são componentes do núcleo comum do *middleware* Ginga, portanto é o trecho no qual o componente P2P estará contido. O aplicativo teste está em *p2p-test-app*. O *middleware* Ginga não permite que aplicações de terceiros sejam iniciadas (o *middleware* deve ser o responsável por iniciá-las).

Com isso a estrutura do diretório de trabalho é:

gingacc–cpp/ implementação do núcleo comum

  gingacc–p2p/ implementação do componente *peer-to-peer*

ginga–cpp/ *main* do middleware

p2p–test–app/ programa de exemplo que usará o componente

3.2.4 Estudo da biblioteca apropriada

Ao longo dos meses de trabalho o grupo de pesquisas se dividiu para escolher o protocolo e bibliotecas a serem utilizadas pelo projeto, esse trabalho não ocorreu de forma linear, algumas decisões foram revistas, e abordagens abandonadas foram retomadas. Consolidada a escolha pelo protocolo XMPP, o trabalho para a utilização da *libPurple* foi caminhando em paralelo ao da utilização da *libJingle*.

3.2.5 Estudo da biblioteca *libJingle*

A biblioteca é mantida no repositório Google Code[2]. A versão da biblioteca utilizada foi 0.4.0. Acompanhando a biblioteca há um programa de exemplo para troca de arquivos, chamado `pcp`, cuja implementação define o uso dos servidores XMMP do GTalk, serviço de chat dos usuários da Google. Por decisão da equipe de desenvolvimento foi mantido como tal, uma vez que o objetivo estava em disponibilizar uma primeira versão do componente funcionando, independentemente da necessidade de se configurar um servidor XMPP para teste, assim sendo, isso não está coberto pelo trabalho.

Após a primeira compilação do programa `pcp` foi observado que o binário possuia, aproximadamente, 14MB. Foi constatado que isso ocorria pois as diretivas de compilação da *libJingle* estavam gerando bibliotecas estáticas. Desse modo, aplicativos que se utilizam disso carregam as bibliotecas em seus binários; porém isso não se adequa ao projeto do *middleware* Ginga, uma vez que aplicativos que quisessem utilizar a biblioteca em questão deveriam contê-la após a compilação. Imaginando um cenário com dezenas de aplicativos, todos com a *libJingle* auto-contida, traria um claro mal uso da mesma, além de não permitir a componentização proposta pelo trabalho. Assim sendo, se fez necessária a alteração das diretivas para a geração de bibliotecas compartilhadas, que estarão no sistema, e que assim sendo, podem ser incluídas como dependência de qualquer aplicação.

3.2.6 Estudo das ferramentas do Sistema de compilação GNU

The GNU Build System, é um conjunto de aplicativos de linha de comando para assistência ao programador que auxiliaa escrita de programas portáveis, rodando em várias plataformas POSIX (Especificações de Interfaces de Sistemas Operacionais). Essas ferramentas são utilizadas pelos desenvolvedores da *libJingle* e também estão no projeto do *middleware* Ginga, portanto foram bastante utilizadas no decorrer deste trabalho.

Autoconf, *Automake* e *Libtool* são as ferramentas do *GNU Build System*. De forma resumida, para compilar projetos que se utilizam desses utilitários são usados os comandos:[3]

A descrição de cada ferramenta de forma sucinta, segundo tradução livre de sua documentação::

Autoconf: ferramenta para produzir *scripts* que configuram automaticamente o código fonte dos pacotes do software a fim de adaptá-los aos diversos tipos de sistemas POSIX.

[2]http://code.google.com/apis/talk/libjingle/index.html — acesso em 25/05/2010
[3]*make install* exige privilégios administrativos do sistema operacional

Automake: ferramenta para gerar automaticamente arquivos *Makefile.in* a partir de arquivos de configuração *Makefile.am*, sendo que este último contém, basicamente, uma série de definições de variáveis, e o arquivo gerado, *Makefile.in*, está de acordo com os padrões *GNU Makefile*.

Libtool: ferramenta que cuida de todos os requisitos para compilar bibliotecas compartilhadas e permitindo a portabilidade delas.

```
$ ./configure
$ make
$ make install
```

Esses comandos implicitamente utilizam as ferramentas do sistema, pois o *script* `configure` gera um arquivo *Makefile*, e o programa `make` compila o projeto a partir do Makefile gerado. Com mais detalhes, o desenvolvedor definiu parâmetros em um arquivo `configure.ac` (anteriomente se utilizava o nome `configure.in`, usado neste projeto inclusive), que é a entrada para o *Autoconf*, gerando então o arquivo `configure`. Esse arquivo é um *script* que tem como entrada um arquivo com o nome de `Makefile.in`, e assim gerar o *Makefile*. Este arquivo `Makefile.in` é a saída da execução do *Autoconf*, este tem como entrada `Makefile.am`, onde o desenvolvedor definiu outros parâmetros para o projeto (a Figura 8 mostra esse fluxo).
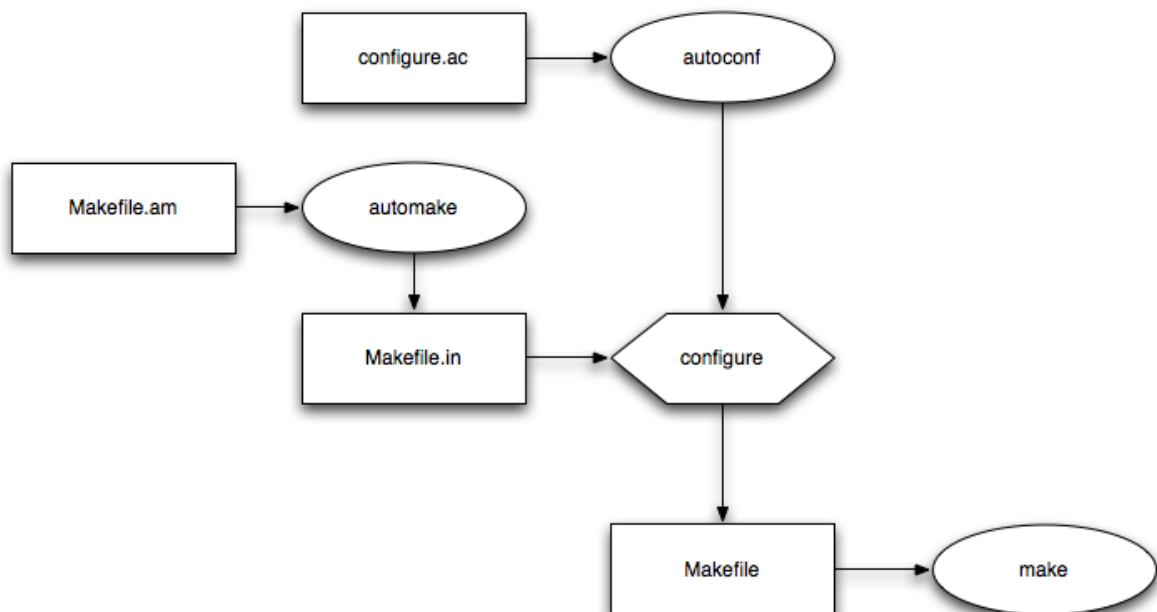
Figura 8: Diagrama de fluxo do Autoconf e Automake

Assim sendo, os arquivos `Makefile.am` e `configure.ac` possuem diretivas que alteram a forma que o projeto será compilado. A ferramenta *make* executa o compilador, *g++*, e o

libtool fica responsável pela geração das bibliotecas compartilhadas ou estáticas. Alguns projetos precisam rodar mais ferramentas antes de serem compilados, e por convenção a chamada para essas execuções ficam no *script* `autogen.sh`. O *middleware* Ginga e a *libJingle* se encaixam nisso e os comandos usados são:

```
$ ./autogen.sh
$ make
$ make install
```

3.2.7 Alteração da *libJingle* para geração de bibliotecas compartilhadas

A SDK (*Software Development Kit*) da *libJingle* possui os seguintes diretórios:[4]

base/ classes para funcionalidades de baixo-nivel como *sockets* e *threads*;

examples/ exemplos de aplicações de chamada de voz e troca de arquivos, sendo essa a usada pelo componente, chamda de 'pcp';

p2p/ classes para negociação, estabelecimento e manutenção das conexões *peer-to-peer*;

session/ classes que especializam o comportamento das classes de `p2p`;

third–party/ diretório onde ficam extensões e biblioteca construídas por terceiros;

xmllite/ *parser* xml; e

xmpp/ classes para envio e recebimento de chamadas XMPP, além de opercações básicas como login.

Como já mencionado a *libJingle*, por definição de seus desenvolvedores, gera apenas bibliotecas estáticos e para alterar isso foram feitas mudanças em todos arquivos `Makefile.am` de sua SDK, fazendo o seguinte:

- ocultou-se os parâmetros que desabilitavam a instalação de bibliotecas;
- alterou-se os parâmetros para a criação dos arquivos `.so`[5] pela *LibTool*.

[4] Diretórios descritos como em sua documentação

[5] Arquivos `.so` são chamados de *shared objects*, binários das bibliotecas compartilhadas

3.2.8 Codificação do componente

A partir do código-fonte da aplicação `pcp` que acompanha a *libJingle* foi construído o componente e a aplicação de exemplo. O código-fonte das mesmas estarão, respectivamente, nos diretórios *gingacc-p2p* e *p2p-test-app*.

Inserido em `ginga-cpp/src/main.cpp` há a chamada para a aplicação teste:

```
P2PTest *p2pTest = P2PTest::getInstance();
p2pTest->start();
```

A Figura 9 mostra o diagrama de classes do componete P2P e da aplicação de teste, um detalhe de implementação é a necessidade da aplicação herdar a classe `::br::pucrio::telemidia::ginga::core::system::thread:: Thread` para que o *middleware* Ginga não trave esperando a execução da mesma.
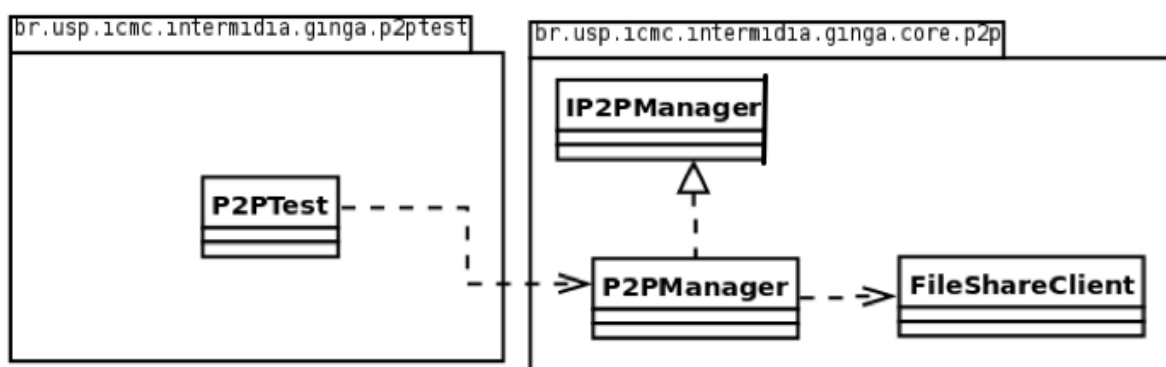
Figura 9: Diagrama de classes do componente e aplicação teste

A classe *P2PTest* possui o método `run()`, sua implementação instancia um objeto da classe *P2PManager* do componente. Essa classe tem o método *connect()*, responsável pela conexão com servidor XMPP e chamada dos métodos implementados na classe *FileShareClient*, responsável por realizar a troca de arquivos.

A classe *P2PTest* é uma estrutura *Singleton*, um padrão de projeto que restringe a criação de apenas uma instância da classe. O código-fonte de *P2PTest* mostra isso no método `getInstance()`, o cronstrutor da classe possuia rotina de *log* usada pelo *middleware* Ginga. O método `run()` configura alguns detalhes para a conexão e em seguida chama o método `connect()` da classe *P2PManager*.

```
P2PTest* P2PTest::_instance = NULL;

P2PTest::P2PTest() {
 logger = LoggerUtil::getLogger(
   "br.usp.icmc.intermidia.ginga.p2ptest.P2PTest");
      LoggerUtil::configure();
```

```
  LoggerUtil_info(logger, "P2PTest::P2PTest()");
  received = false;
 }

 P2PTest* P2PTest::getInstance() {
  if (_instance == NULL) {
   _instance = new P2PTest();
  }
  return _instance;
 }

 P2PTest::~P2PTest() {
 }

 bool P2PTest::isReceived() {
  return received;
 }

 void P2PTest::run() {
  LoggerUtil_info(logger, "void P2PTest::run()");
//  cout << "void P2PTest::start()" << endl;

  talk_base::InsecureCryptStringImpl pass;
  std::string& senha = pass.password();
  senha = "0987)(*&";

  P2PManager *p2p = P2PManager::getInstance();
  p2p->connect("talk.google.com", 5222, "diogodecpedrosa@gmail.com", pass);
  LoggerUtil_info(logger, "antes de unlockConditionSatisfied");
  received = true;
  unlockConditionSatisfied();
  LoggerUtil_info(logger, "depois de unlockConditionSatisfied");
 }
```

Código-fonte de *P2PTest.cpp*

P2PManager também é uma classe *Singleton*, seu método `connect()` é responsável por criar a conexão XMPP e preparar parâmetros para a instância de *FileShareClient*.

```
 P2PManager* P2PManager::_instance = NULL;

#if HAVE_COMPSUPPORT
 static IComponentManager* cm = IComponentManager::getCMInstance();
#endif

 P2PManager::P2PManager() {
```

```
 logger = LoggerUtil::getLogger(
   "br.usp.icmc.intermidia.ginga.core.p2p.P2PManager");
       LoggerUtil::configure();
 LoggerUtil_info(logger, "Construtor");

 this->isConnected = false;
}

P2PManager::~P2PManager() {
 delete _instance;
}

P2PManager* P2PManager::getInstance() {
 if (_instance == NULL) {
  _instance = new P2PManager();
 }
 return _instance;

}

void P2PManager::connect(string server, int port,
  string username, talk_base::InsecureCryptStringImpl pass) {

 this->isConnected = true;

 LoggerUtil_info(logger, "connect(server, port, username, pass)");

 talk_base::LogMessage::LogToDebug(talk_base::LS_ERROR + 1);
 talk_base::InitializeSSL();

 // Checa se o username passado possui um formato v lido.
 buzz::Jid jid = buzz::Jid(username);
 if (!jid.IsValid() || jid.node() == "") {
  LoggerUtil_info(logger,
    "Invalid JID. JIDs should be in the form user@domain");
  return;
 }

 // Define os valores de um XmppClientSettings
 buzz::XmppClientSettings xcs;
 xcs.set_user(jid.node());
 xcs.set_resource("P2PManager");
 xcs.set_host(jid.domain());
 xcs.set_use_tls(true);
 xcs.set_pass(talk_base::CryptString(pass));
 xcs.set_server(talk_base::SocketAddress(server, port));
```

```
  // Cria uma nova thread e define a como ativa
  talk_base::PhysicalSocketServer ss;
  talk_base::Thread main_thread(&ss);
  talk_base::ThreadManager::SetCurrent(&main_thread);

  // Prepara os par metros e cria uma inst ncia do FileShareClient
  XmppPump pump;
  buzz::Jid j = buzz::JID_EMPTY;
  cricket::FileShareManifest *manifest = new cricket::FileShareManifest();
  char cwd[256];
  getcwd(cwd, sizeof(cwd));
  FileShareClient fs_client(pump.client(), j, manifest, cwd);

  // Define que a fun  o OnStateChange do cliente de compartilhamento de
  // arquivo deve ser chamada quando algum evento de mudan a de estado
  // ocorrer.
  pump.client()->SignalStateChange.connect(&fs_client,
    &FileShareClient::OnStateChange);

  // Realiza o login,  espera o recebimento de um arquivo e desconecta.
  pump.DoLogin(xcs, new XmppSocket(true), NULL);
  main_thread.Run();
  pump.DoDisconnect();
 }


extern "C" ::br::usp::icmc::intermidia::ginga::core::p2p::IP2PManager*
  createP2PManager() {

 return ::br::usp::icmc::intermidia::ginga::core::p2p::
   P2PManager::getInstance();
}

extern "C" void destroyP2PManager(
  ::br::usp::icmc::intermidia::ginga::core::p2p::IP2PManager* p2pm) {

 delete p2pm;
}
```

Código-fonte de *P2PManager.cpp*

A classe *FileShareClient* implementá a extensão Jingle, a restruturação para a ampliação da API do componente se dará a partir desta classe, que concentra todo o serviço de troca de arquivos.

```

  FileShareClient::FileShareClient(buzz::XmppClient *xmppclient, const buzz
```

```
    ::Jid &send_to,
  const cricket::FileShareManifest *manifest, std::string root_dir) :
  xmpp_client_(xmppclient),
  root_dir_(root_dir),
  send_to_jid_(send_to),
  waiting_for_file_(send_to == buzz::JID_EMPTY),
  manifest_(manifest) {

 cout << "Construtor de FileShareClient" << endl;
}

void FileShareClient::OnStateChange(buzz::XmppEngine::State state) {
 cout << "OnStateChange" << endl;
  switch (state) {
  case buzz::XmppEngine::STATE_START:
    std::cout << "Connecting..." << std::endl;
    break;
  case buzz::XmppEngine::STATE_OPENING:
    std::cout << "Logging in. " << std::endl;
    break;
  case buzz::XmppEngine::STATE_OPEN:
    std::cout << "Logged in as " << xmpp_client_->jid().Str() << std::
        endl;
    if (!waiting_for_file_)
      std::cout << "Waiting for " << send_to_jid_.Str() << std::endl;
    OnSignon();
    break;
  case buzz::XmppEngine::STATE_CLOSED:
    std::cout << "Logged out." << std::endl;
    break;
  }
}

void FileShareClient::OnJingleInfo(const std::string & relay_token,
                   const std::vector<std::string> &relay_addresses,
                   const std::vector<talk_base::SocketAddress> &
                      stun_addresses) {
 cout << "OnJingleInfo" << endl;
  port_allocator_->SetStunHosts(stun_addresses);
  port_allocator_->SetRelayHosts(relay_addresses);
  port_allocator_->SetRelayToken(relay_token);
}


void FileShareClient::OnStatusUpdate(const buzz::Status &status) {
 cout << "OnStatusUpdate" << endl;
  if (status.available() && status.fileshare_capability()) {
```

```
      // A contact's status has changed. If the person we're looking for is
           online and able to receive
      // files, send it.
      if (send_to_jid_.BareEquals(status.jid())) {
std::cout << send_to_jid_.Str() << " has signed on." << std::endl;
cricket::FileShareSession* share = file_share_session_client_->
    CreateFileShareSession();
share->Share(status.jid(), const_cast<cricket::FileShareManifest*>(
    manifest_));
send_to_jid_ = buzz::Jid("");
      }

   }
 }

 void FileShareClient::OnMessage(talk_base::Message *m) {
  cout << "OnMessage" << endl;
   ASSERT(m->message_id == MSG_STOP);
   talk_base::Thread *thread = talk_base::ThreadManager::CurrentThread();
   delete session_;
   thread->Stop();
 }

 std::string FileShareClient::filesize_to_string(unsigned int size) {
  cout << "filesize_to_string" << endl;
   double size_display;
   std::string format;
   std::stringstream ret;

   // the comparisons to 1000 * (2^(n10)) are intentional
   // it's so you don't see something like "1023 bytes",
   // instead you'll see ".9 KB"

   if (size < 1000) {
     format = "Bytes";
     size_display = size;
   } else if (size < 1000 * 1024) {
     format = "KiB";
     size_display = (double)size / 1024.0;
   } else if (size < 1000 * 1024 * 1024) {
     format = "MiB";
     size_display = (double)size / (1024.0 * 1024.0);
   } else {
     format = "GiB";
     size_display = (double)size / (1024.0 * 1024.0 * 1024.0);
   }
```

```
  ret << std::setprecision(1) << std::setiosflags(std::ios::fixed) <<
      size_display << " " << format;
  return ret.str();
}

void FileShareClient::OnSessionState(cricket::FileShareState state) {
 cout << "OnSessionState" << endl;
  talk_base::Thread *thread = talk_base::ThreadManager::CurrentThread();
  std::stringstream manifest_description;

  switch(state) {
  case cricket::FS_OFFER:

    // The offer has been made; print a summary of it and, if it's an
        incoming transfer, accept it

    if (manifest_->size() == 1)
      manifest_description <<  session_->manifest()->item(0).name;
    else if (session_->manifest()->GetFileCount() && session_->manifest()
        ->GetFolderCount())
      manifest_description <<  session_->manifest()->GetFileCount() << "
          files and " <<
               session_->manifest()->GetFolderCount() << " directories";
    else if (session_->manifest()->GetFileCount() > 0)
      manifest_description <<  session_->manifest()->GetFileCount() << "
          files";
    else
      manifest_description <<  session_->manifest()->GetFolderCount() <<
          " directories";

    size_t filesize;
    if (!session_->GetTotalSize(filesize)) {
      manifest_description << " (Unknown size)";
    } else {
      manifest_description << " (" << filesize_to_string(filesize) << ")"
          ;
    }
    if (session_->is_sender()) {
      std::cout << "Offering " << manifest_description.str()  << " to "
          << send_to_jid_.Str() << std::endl;
    } else if (waiting_for_file_) {
std::cout << "Receiving " << manifest_description.str() << " from " <<
   session_->jid().BareJid().Str() << std::endl;
session_->Accept();
waiting_for_file_ = false;
```

```
    }
    break;
  case cricket::FS_TRANSFER:
    std::cout << "File transfer started." << std::endl;
    break;
  case cricket::FS_COMPLETE:
    thread->Post(this, MSG_STOP);
    std::cout << std::endl << "File transfer completed." << std::endl;
    break;
  case cricket::FS_LOCAL_CANCEL:
  case cricket::FS_REMOTE_CANCEL:
    std::cout << std::endl << "File transfer cancelled." << std::endl;
    thread->Post(this, MSG_STOP);
    break;
  case cricket::FS_FAILURE:
    std::cout << std::endl << "File transfer failed." << std::endl;
    thread->Post(this, MSG_STOP);
    break;
  }
}

void FileShareClient::OnUpdateProgress(cricket::FileShareSession *sess) {
  // Progress has occured on the transfer; update the UI
  cout << "OnUpdateProgress" << endl;
  size_t totalsize, progress;
  std::string itemname;
  unsigned int width = 79;

  struct winsize ws;
  if ((ioctl(STDOUT_FILENO, TIOCGWINSZ, &ws) == 0))
    width = ws.ws_col;

  if(sess->GetTotalSize(totalsize) && sess->GetProgress(progress) && sess
     ->GetCurrentItemName(&itemname)) {
    float percent = (float)progress / totalsize;
    unsigned int progressbar_width = (width * 4) / 5;

    const char *filename = itemname.c_str();
    std::cout.put('\r');
    for (unsigned int l = 0; l < width; l++) {
      if (l < percent * progressbar_width)
 std::cout.put('#');
     else if (l > progressbar_width && l < progressbar_width + 1 + strlen
        (filename))
       std::cout.put(filename[l-(progressbar_width + 1)]);
     else
       std::cout.put(' ');
```

```
      }
      std::cout.flush();
    }
}

void FileShareClient::OnResampleImage(std::string path, int width, int
    height, talk_base::HttpTransaction *trans) {
 cout << "OnResampleImagem" << endl;

  talk_base::FileStream *s = new talk_base::FileStream();
  if (s->Open(path.c_str(), "rb"))
    session_->ResampleComplete(s, trans, true);
  else {
    delete s;
    session_->ResampleComplete(NULL, trans, false);
  }
}

void FileShareClient::OnFileShareSessionCreate(cricket::FileShareSession
    *sess) {
 cout << "OnFileShareSessionCreate" << endl;

  session_ = sess;
  sess->SignalState.connect(this, &FileShareClient::OnSessionState);
  sess->SignalNextFile.connect(this, &FileShareClient::OnUpdateProgress);
  sess->SignalUpdateProgress.connect(this, &FileShareClient::
      OnUpdateProgress);
  sess->SignalResampleImage.connect(this, &FileShareClient::
      OnResampleImage);
  sess->SetLocalFolder(root_dir_);
}

void FileShareClient::OnSignon() {
 cout << "OnSignon" << endl;
  std::string client_unique = xmpp_client_->jid().Str();
  cricket::InitRandom(client_unique.c_str(), client_unique.size());

  buzz::PresencePushTask *presence_push_ = new buzz::PresencePushTask(
      xmpp_client_);
  presence_push_->SignalStatusUpdate.connect(this, &FileShareClient::
      OnStatusUpdate);
  presence_push_->Start();

  buzz::Status my_status;
  my_status.set_jid(xmpp_client_->jid());
  my_status.set_available(true);
  my_status.set_show(buzz::Status::SHOW_ONLINE);
```

```
  my_status.set_priority(0);
  my_status.set_know_capabilities(true);
  my_status.set_fileshare_capability(true);
  my_status.set_is_google_client(true);
  my_status.set_version("1.0.0.66");

  buzz::PresenceOutTask* presence_out_ =
    new buzz::PresenceOutTask(xmpp_client_);
  presence_out_->Send(my_status);
  presence_out_->Start();

  port_allocator_.reset(new cricket::HttpPortAllocator(&network_manager_,
      "pcp"));

  session_manager_.reset(new cricket::SessionManager(port_allocator_.get
      (), NULL));

  cricket::SessionManagerTask * session_manager_task = new cricket::
      SessionManagerTask(xmpp_client_, session_manager_.get());
  session_manager_task->EnableOutgoingMessages();
  session_manager_task->Start();

  buzz::JingleInfoTask *jingle_info_task = new buzz::JingleInfoTask(
      xmpp_client_);
  jingle_info_task->RefreshJingleInfoNow();
  jingle_info_task->SignalJingleInfo.connect(this, &FileShareClient::
      OnJingleInfo);
  jingle_info_task->Start();

  file_share_session_client_.reset(new cricket::FileShareSessionClient(
      session_manager_.get(), xmpp_client_->jid(), "pcp"));
  file_share_session_client_->SignalFileShareSessionCreate.connect(this,
      &FileShareClient::OnFileShareSessionCreate);
  session_manager_->AddClient(NS_GOOGLE_SHARE, file_share_session_client_
      .get());
}
```

Código-fonte de *FileShareClient.cpp*

3.3 Resultados Obtidos

O aplicativo `pcp`, parte da SDK da *libJingle*, foi compilado em outro computador e configurado com uma conta XMMP diferente da aplicação de teste do componente, dentro do *middleware* para a demostração de envio de um arquivo deste cliente até o *middleware*

Ginga.

A execução do binário `ginga` imprime no terminal da máquina virtual *logs*, que foram utilizados pra confirmar a execução da aplicação, e a partir disso foi executada em outro computador o aplicativo pcp, após o estabelecimento da conexão entre os dois pontos o arquivo foi enviado com sucesso.

O diagrama da Figura 10 mostra o *middleware* Ginga conectado ao servidor *GTalk*, assim como o aplicativo de troca de arquivos pcp na máquina *Ubuntu*. A troca de arquivos é feita diretamente entre os pontos, o arquivo trocado não é enviado ao servidor.
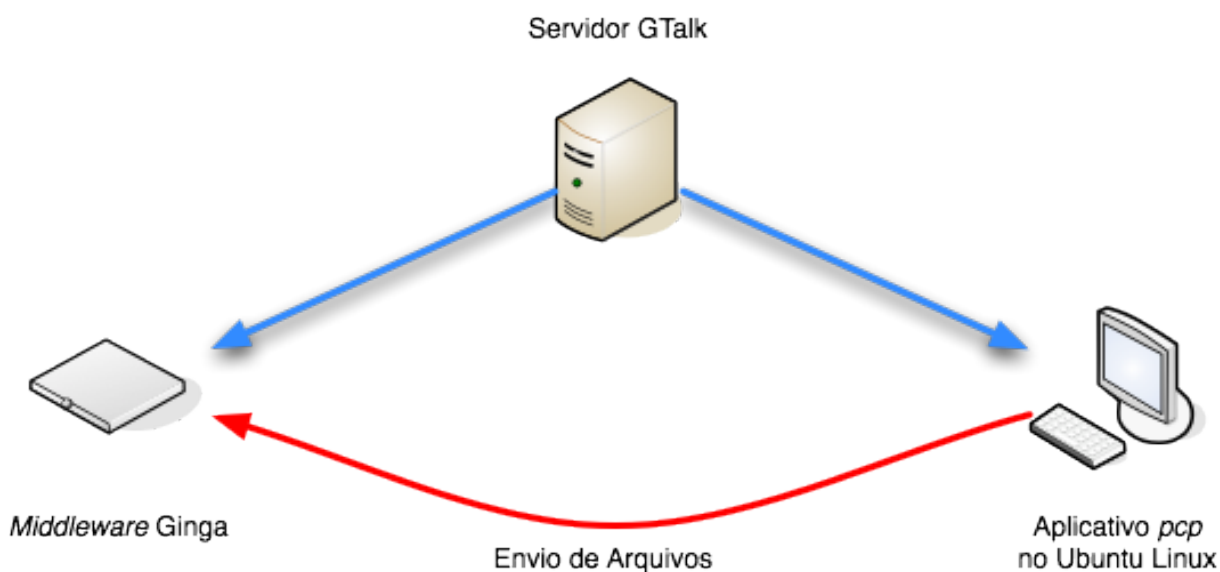
Figura 10: Diagrama do teste final do componente

Caberá a próxima versão realizar o sentido contrário do exibido pelo diagrama, ou seja, o envio de um arquivo a partir do *middleware* Ginga.

A versão final de todo código-fonte gerado ou modificado por e para este trabalho está disponível em http://www.armandoneto.com/edu/.

3.4 Dificuldades e Limitações

Um fator que alterou a abordagem do trabalho foi o tempo disponível, pois em outra situação a escolha da biblioteca utilizada teria sido diferente. A *libPurple* é bastante completa e poderia oferecer um serviço mais completo e dentro da especificação atual da extensão do protocolo, porém em contra-partida traria uma complexidade dispensável ao componente.

Como mencionado anteriormente, servidores do *GTalk* foram usados para a demonstração, e a implementação atual inclui essa utilização. Faz-se necessário experimentar a

utilização da biblioteca fora desse ambiente, pelo fato de a biblioteca *libJingle* ser fornecida pela *Google Inc.*

O trabalho se utilizou de software de código aberto, o que significa que centenas de programadores trabalharam para o desenvolvimento do de todo o material utilizado, e a experiência desses programadores transforma a tarefa de acompanhar o andamento do projeto e o entendimento de seu trabalho numa tarefa bastante custosa, isso se refletiu no desenvolvimento deste trabalho.

Um gasto de tempo enorme ocorreu, pois ora o projeto optava por trabalhar com uma biblioteca, ora por outra. Tudo esse trabalho foi realizado com a consciência de que a primeira versão (detalhada neste trabalho) não precisaria ter todas as funcionalidades, mas que as determinações tomadas fossem pertinentes para levar o projeto a próxima versão, e com isso minimizar os problemas de decisão futuros.

3.5 Considerações Finais

Este capítulo apresentou detalhes do desenvolvimento do componente proposto a partir das ferramentas e bibliotecas de apoio selecionadas pelo grupo de trabalho. O componente foi testado com uma aplicação que demonstrou, com sucesso, a troca de um arquivo a partir de uma máquina rodando o *Ubuntu Linux*. No próximo capítulo é apresentada a conclusão sobre o trabalho realizado.

4 *Conclusão*

4.1 Contribuições

O trabalho contribuiu para a inserção de novos recursos no âmbito da TV Digital no Brasil, o que ajudará no amadurecimento do *middleware* Ginga. O trabalho serviu também como mais um experimento da evolução que a televisão deve sofrer ao decorrer dos próximos anos, ao ser tratada como uma nova plataforma de interatividade – o que é relevante por ser a TV, atualmente, a forma dominante para acesso a informação no Brasil.

Outro ponto foi o amadurecimento do aluno em relação ao desenvolvimento na linguagem C++ com as ferramentas GNU, além de novos conceitos adquiridos sobre bibliotecas dinâmicas de software, licenças e a importância do software de código-aberto para o desenvolvimento da pesquisa.

Sobretudo, a experiência de realizar um trabalho nos laboratórios da universidade ampliou o olhar do aluno sobre o meio acadêmico, e isto traz um benefício que não pode ser medido, além de ter sido o primeiro contato do aluno com um projeto de software real.

4.2 Considerações sobre o Curso de Graduação

O curso Bacharelado em Informática proveu o conhecimento necessário para a execução deste trabalho. Em particular, a ênfase em Administração e Gerenciamento de Redes serviu de grande complemento, o que possibilitou o entendimento de termos e noções necessários durante o desenvolvimento do projeto.

Outro ponto a ser destacado é a facilidade de encontrar artigos científicos que enriqueceram o projeto, uma vez que a Universidade de São Paulo provê acesso às bibliotecas digitais ACM (Association for Computing Machinery) e IEEE (Institute of Electrical and Electronics Engineers), utilizadas para consulta durante o projeto.

Um ponto fraco é a exigência de que os alunos do curso devem confeccionar, com o

formalismo cabível, uma monografia, sendo que ao longo dos anos não houve o exercício de tal experiência, salvo aqueles que realizaram Iniciação científica. Os trabalhos realizados no contexto das disciplinas do curso não são suficientes, na opinião do aluno, para compensar a falta do exercício supra citado.

Outro ponto a ser levado em consideração é o fato do projeto necessitar estudo complementar em vários tópicos, o que toma certo tempo, e esse estudo acaba encaminhando o desenvolvimento do trabalho para algo que pode não ser o objetivo descrito no título do trabalho, e a exigência do título na matrícula da disciplina Projeto de Graduação não prevê tal acontecimento.

4.3 Trabalhos Futuros

Independentemente do envolvimento do autor, o trabalho com o *middleware* Ginga deve ter continuidade. Nesta versão o *middleware* apenas recebe arquivos, a próxima versão deve ser capaz de suportar o envio. O componente P2P deverá ter adicionado em suas funcionalidades os recursos para troca de mensagens e independência com o servidor XMPP da Google, além da ampliação de sua API.

Referências

COHEN, B. *The BitTorrent Protocol Specification*. 2009. *On-line*. Disponível em: <http://www.bittorrent.org/beps/bepne 0003.html>. Acesso em: 14 mar. 2010.

FREE SOFTWARE FOUNDATION, INC. *Autoconf Documentation*. [S.l.]. Disponível em: <http://www.gnu.org/software/autoconf/manual/autoconf.html>. Acesso em: 27 mai. 2010.

FREE SOFTWARE FOUNDATION, INC. *Automake Documentation*. [S.l.]. Disponível em: <http://www.gnu.org/software/automake/manual/automake.html>. Acesso em: 27 mai. 2010.

FREE SOFTWARE FOUNDATION, INC. *Libtool Dcumentation*. [S.l.]. Disponível em: <http://www.gnu.org/software/libtool/manual/libtool.html>. Acesso em: 27 mai. 2010.

GHODSI, A. *Distributed k-ary System: Algorithms for Distributed Hash Tables*. Tese (PhD Dissertation) — KTH—Royal Institute of Technology, Stockholm, Sweden, out. 2006.

GOOGLE, INC. *libjingle Developer Guide*. [S.l.], 2009. Disponível em: <http://code-.google.com/apis/talk/libjingle/index.html>. Acesso em: 27 mai. 2010.

LUDWIG, S. et al. *XEP-0166: Jingle*. 2009. *On-line*. Disponível em: <http://xmpp.org-/extensions/xep-0166.html>. Acesso em: 24 abr. 2010.

MONTEZ, C.; BECKER, V. *TV Digital Interativa: Conceitos, Desafios e Perspectivas para o Brasil*. 2a ed.. ed. [S.l.]: UFSC, 2006.

SAINT-ANDR.; SMITH, K.; TRON, R. *XMPP: The Definitive Guide: Building Real-Time Applications with Jabber Technologies*. [S.l.]: O'Reilly Media, Inc., 2009.

SCHOLLMEIER, R. A definition of peer-to-peer networking for the classification of peer-to-peer architectures and applications. p. 101 –102, aug 2001.

SOARES, L. F. G. *GingaFrEvo & GingaRAP – Evolução do Middleware Ginga para Múltiplas Plataformas (Componentização) e Ferramentas para Desenvolvimento e Distribuição de Aplicações Declarativas*. 2009. *Edital CTIC*.

SOARES, L. F. G. *Nested Context Language 3.0 Part 11 - Declarative Objects in NCL: Nesting Objects with NCL Code in NCL Documents*. 2009. *On-line*. Disponível em: <http://www.ncl.org.br/documentos/NCL3.0-DO.pdf>. Acesso em: 30 abr. 2010.